Abstracting the Template Instantiation Relation in C++

Andrew Sutton, Ryan Holeman, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{asutton, rholeman, jmaletic}@cs.kent.edu

Abstract

A source code model that supports the static analysis of C++ templates and template metaprograms is presented. Analogous to techniques for object-oriented and procedural software (e.g., the abstraction of call graphs, inheritance hierarchies, etc.), this model provides a basis for maintenance concerns such as program comprehension, fact extraction, and impact analysis of generic code. The source code model is used to derive the template instantiation graph, and potential applications of this model discussed. An application to reverse engineer this model from source code is described.

1. Introduction

Generic programming is an increasingly accepted programming paradigm for the creation of highly adaptable, high performance, and safety-critical software libraries. This is clearly demonstrated by the widespread usage of the C++ STL and the Boost C++ Libraries. The increasing viability of generic programming as a dominant paradigm is evidenced by the introduction of generics to Java and C#. In C++ generic programming is predicated on templates.

Template metaprogramming is a set of template programming techniques and idioms that use the compiler's built-in type system and partial evaluation capabilities to write compile-time programs that operate on types or integral expressions. Template metaprograms are used frequently in generic libraries to programmatically maintain generic abstractions over diverse kinds of types. Unfortunately, the use of this programming paradigm and associated techniques has drastically outpaced methods and tools to support the analysis and understanding of generic source code.

Currently, there are no standard methods or techniques to statically analyze generic libraries. One could instantiate the generic source code and perform analyses on its concrete representation, but this is known to be inadequate for the analysis of abstract, generic concepts embodied within the code since the approach necessarily elides the original definition. The lack of analytic support for generic source code has resulted in a substantial gap between our ability to construct generic libraries and our ability to maintain them.

In this paper, we take a step towards providing support for maintenance concerns of generic libraries and template metaprograms. We describe a program model designed to support the static analysis of generic source code in C++. From this model, we derive the template instantiation graph, which we posit could be used much like a call graph to support maintenance applications such as impact analysis, feature location, or improving program comprehension tools. The template model and graph also support *variadic templates*, a new language feature appearing in C++0x.

To support this work, we have extended our software-modeling framework, *srctools*, to support the reverse engineering of templates. Examples in this paper are taken from the Origin C++0x Libraries[1], a repository for experimentation with generic library design.

2. Related Work

Static program analysis and the recovery of source code models is the cornerstone of a wide variety of applications that support software maintenance and evolution tasks. However, there is very little work adapting common approaches in the presence of templates. Control flow graphs are used to support static checking of STL usage [1]. A dependence graph that maps expressions to C++0x concepts is used to support impact analysis for the modification of concept definitions [2, 3]. A similar approach is used in the inference of type constraints in function templates [4].

The method of extracting source code models can range from lightweight, lexical approaches [5] to heavyweight parsing and analysis [6-9], to post-translation IR, bytecode, or metadata analysis [10, 11]. In general, it is well known that support for the reverse engineering and modeling of template source code is insufficient.

---

[1] See http://www.sdml.info/projects/origin/ for details.

3. A Source Code Model for Templates

In order to support applications that reason about the structure of template source code, template metaprograms, and generic libraries, we must first provide a means of modeling the program structure and semantics. We have done so by creating a relatively simple source code model to support the reverse engineering of these elements directly from the original source code. Since the intent of the model is to support the recovery and abstraction of program information from C++ source code, strict adherence to the structure and semantics of the C++ language is not critical.

At the core of this model is the representation of templates and their relationships. Figure 1 shows a UML description of the program model for templates and template specialization. In our model, a template is any parameterized, scoped element such as classes or functions, and member functions.

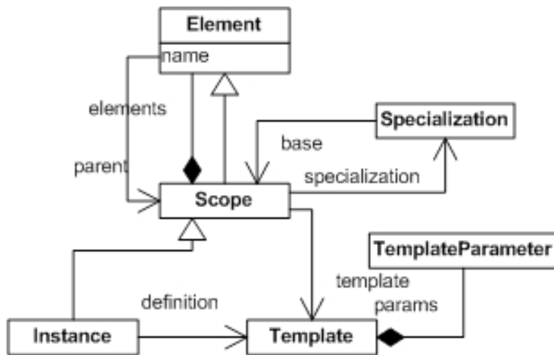

Figure 1. A subset of the template source code model that defines and relates template and template specialization elements

Each template scope contains a sequence of template parameters. Template parameters describe the kinds of elements over which the template is parameterized. C++ allows the definition of both explicit and partial specializations. Template specialization allows the writing of alternative class or function template implementations that are matched by a pattern of expressions when the template is instantiated. Explicit specialization is the matching of a template instance to a concrete set of arguments, whereas partial specialization uses template parameters to act as "variables" or placeholders within the matched pattern. If specialized, the template element will refer to a specialization element that stores the pattern of arguments in explicit or partial specialization.

A template instance represents the instantiation of a template element and can denote either a class template instance or function template instance. Each instance refers to its template definition.

There are three kinds of template parameters: type parameters, non-type (or integral) parameters, and template template parameters. These are shown in Figure 2. Each kind of parameter is derived from a model element that best represents its semantics within a template scope.

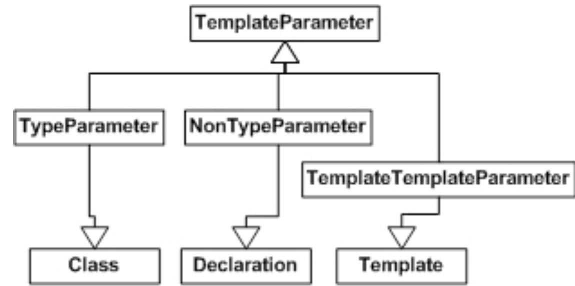

Figure 2. A subset of the C++ program model that describes generic program elements.

One aspect of this model that is not shown in the UML models is the representation of the names of template elements. The name of every template and specialized element is appended by its specialization pattern, and the names of template parameters are "cannibalized" according to their kind and index. For example the std::vector template would be named `vector<$T1,$T2>`, and its partial specialization on `bool` would be named `vector<bool,$T1>`.

This model, although somewhat simple, provide a solid foundation for abstracting information about the templates, their instances, and the way they are used within generic libraries and template metaprograms. In the next section, we describe one such abstraction and discuss its potential use in software maintenance applications.

4. The Instantiation Graph

One of the most challenging analytical problems caused by the pervasive use of templates within a program is their effective decoupling of program elements. Since type information is only propagated through templates during instantiation, the accurate recovery and description of program design from the source code is made very difficult. To better facilitate reverse engineering of such programs, we define the template instantiation graph.

The *instantiation graph* relates scoped elements to the templates that they instantiate. Formally, we define the instantiation graph as a directed multigraph $IG = (V,E)$. Each vertex is labeled with a class template name. A vertex u is connected to a vertex v if u refers to a type that is an instantiation of the template labeled by v. The edge (u, v) is labeled with the sequence of arguments supplied to the template instantiation.

If a template is found to have any specializations, then a *specialization vertex* is created and labeled with the template name, but not the arguments. Instantiations of

specialized template are connected to specialization vertex rather than the concrete template.

```
template <typename P> struct extract_policies {
  typedef typename P::type type;
};
template <typename... P> struct wrap_policies {
  typedef policies<P...> type;
};
template <typename... P> struct wrap_policies<
  policies<P...>> {
  typedef policies<P...> type;
};
template <typename P> struct normalize_policy {
  typedef typename mp::eval_if<
    typename is_derived_policy<P>::type,
    extract_policies<P>,
    wrap_policies<P>>::type type
};
template <typename T, typename P>
class vector<T, P> : public vector_base<
    T, typename normalize_policy<P>::type
  > { };
```

Figure 3. The the `extract`, `wrap`, and `normalize` metafunctions are used to determine the instantiation of `vector_base` for the given `vector` specialization.

For example, consider the code shown in Figure 3. In this program, the `vector` class template specialization normalizes the template parameter `P` to ensure that the class is wrapped in a `policies` template. The resulting instantiation graph is given in Figure 2. Here, the instantiations of the arguments to `eval_if` metafunction are attached to the `normalize_policy` vertex. The `wrap_policy` vertex represents optional instantiation of either specialization (not unlike virtual functions in polymorphic classes).

Instantiations of nested class templates require special consideration. Consider an abbreviated version of the `rebind_allocator` metafunction shown in Figure 5.

```
template <typename T, typename P>
struct rebind_allocator {
  typedef typename extract_allocator<P>::type A;
  typedef A::template rebind<T>::other type;
};
```

Figure 5. The `rebind_allocator` metafunction uses the `extract` metafunction to retrieve an allocator type, and then `rebind`s it to the type `T`.

The metafunction uses a `typedef` to create a *type variable*. In our analysis we treat type variables as first-class generic program elements that can behave as classes or as class templates. The instantiation graph corresponding to this metafunction is given in Figure 6.

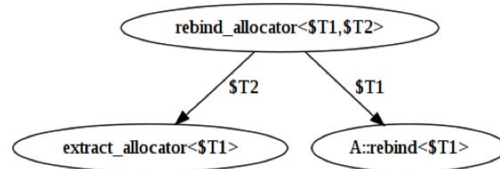

Figure 6. The instantiation graph of the `rebind_allocator` metafunction instantiates both `extract_allocator` and the `rebind` template of the type variable `A`.

Beyond its obvious applications as a visual aid to program comprehension, the template instantiation graph will be invaluable to analyses that support metafunction debugging [12]. For example, a variant of this graph could be defined to model control flow for template

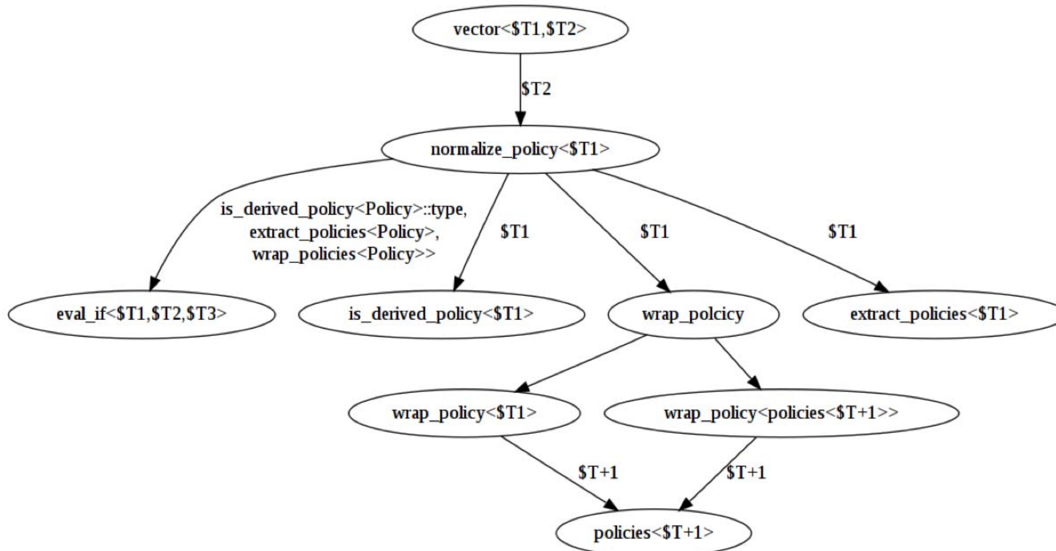

Figure 4. The template instantiation graph shows the dependencies between classes on templates.

metaprograms by attaching special semantics to selective (e.g., `eval_if`) or recursive templates. This graph could also be used to emulate data flow for template parameters. Control flow and data flow analysis are already widely used in a variety of software maintenance applications including impact analysis, program slicing, and feature location. Modeling the abstract structure of template metafunctions will provide similar facilities for reasoning about generic source code.

5. Implementation

We have implemented the reverse engineering and modeling of templates and their instances by extended the reverse engineering capabilities of the *srctools* framework and application suite[2]. The *srctools* project is built on top of srcML[3], an XML markup for source code [13].

The *srctools* framework enables the construction of advanced reverse engineering and program analysis applications by providing hooks (via signals and slots) that correspond to specific parsing events such as the inclusion of a file or the definition of a class. The *srcfacts* application (part of *srctools*) is a C++ fact extractor that outputs program information and abstractions into a relational database. The instantiation graph is trivially reconstructed from this database and rendered using Graphviz.

6. Conclusions and Future Work

We described a program model for reverse engineering elements of C++ generic libraries and template metaprograms. We derive the template instantiation graph from this model, and discuss potential applications of this model in the software maintenance area. In the future, we plan to continue the development of these models and abstractions to support new features for generic programming appearing in C++0x (e.g., variadic templates and concepts) and use these techniques to target specific maintenance applications.

Despite the increasing volume of generic and template software, the advancement software engineering technologies to support the generic programming paradigm have lagged behind its rate of adoption. This work is a first step in addressing this gap.

7. References

[1] D. Gregor and S. Schupp, "STLlint: Lifting Static Checking from Languages to Libraries," Software: Practice and Experience, vol. 36, pp. 225-254, Mar 2005, 2005.

[2] M. Zalewski and S. Schupp, "Changing Iterators with Confidence: A Case Study of Change Impact Analysis Applied to Conceptual Specifications," in Workshop on Library-Centric Software Design (LCSD'05), San Diego, CA, Oct 16, 2005, pp. 64-74.

[3] M. Zalewski and S. Schupp, "Change Impact Analysis for Generic Libraries," in 22nd International Conference on Software Maintenance (ICSM'06), Philadelphia, PA, Sep 24-27, 2006, pp. 35-44.

[4] A. Sutton and J. I. Maletic, "Automatically Identifying C++0x Concepts in Function Templates," in 24th International Conference on Software Maintenance (ICSM'04), Beijing, China, Sep 28-Oct 4, 2008, pp. 57-66.

[5] G. Murphy and D. Notkin, "Lightweight Lexical Source Model Extraction," ACM Transactions on Software Engineering and Methodology, vol. 5, pp. 262-292, Jul 1996, 1996.

[6] Y.-F. Chen, M. Nishimoto, and C. V. Ramamoorthy, "The C Information Abstraction System," IEEE Transactions on Software Engineeering, vol. 16, pp. 325-334, Mar 1990, 1990.

[7] T. Dean, A. Malton, and R. Holt, "Union Schemas as a Basis for a C++ Extractor," in 8th Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, Oct 2-5, 2001, pp. 59-70.

[8] R. Ferenc, F. Magyar, Á. Beszédes, A. Kiss, and M. Tarkiainen, "Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems," in 6th Symposium on Programming Languages and Software Tools (SPLST'01), Szeged, Hungary, Jun, 2002, pp. 16-27.

[9] R. Ferenc, I. Siket, and T. Gyimóthy, "Extracting Facts from Open Source Software," in 20th International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, Sep 11 - 14, 2004, pp. 60-69.

[10] T. Gshwind, M. Pinzger, and H. Gall, "TUAnalyzer - Analyzing Templates in C++ Code," in 11th Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, Nov 8-12, 2004, pp. 48-57.

[11] N. Kraft, B. Malloy, and J. Power, "A Tool Chain for Reverse Engineering C++ Applications," Science of Computer Programming, vol. 69, pp. 3-13, Dec 2007, 2007.

[12] Z. Porkoláb, J. Mihalicza, and Á. Sipos, "Debugging C++ Template Metaprograms," in 5th International Conference on Generative Programming and Component Engineering (GPCE'06), Portland, Oregon, Oct 22-26, 2006, pp. 255-264.

[13] J. I. Maletic, M. L. Collard, and A. Marcus, "Source Code Files as Structured Documents," in 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, Jun 27-29, 2002, pp. 289-292.

[2] See http://www.sdml.info/projects/srctools for more information.
[3] See http://www.sdml.info/projects/srcML for more information.